CONSTRAINT LOGIC PROGRAMMING FOR REASONING ABOUT DISCRETE EVENT PROCESSES*

J. S. OSTROFF

▷ The purpose of this paper is to show that *constraint logic programming* is a useful computational logic for modeling, simulating, and verifying real-time discrete event processes. The designer's knowledge about discrete event processes can be represented by a constraint logic program in a fashion that stays close to the mathematical definition of the processes, and can be used to semiautomate verification of possibly infinite-state systems. The constraint language CPL($\Re$) is used to illustrate verification techniques. ◁

1. INTRODUCTION

The purpose of this paper is to show that *constraint logic programming* is a useful computational logic for modeling, simulating, and verifying real-time discrete event processes. The constraint logic language CLP($\Re$) reported in [11], will be used to illustrate the main ideas. The rest of this introduction discusses the nature of real-time discrete event processes, and the use of constraint logic programming for them.

1.1. Timed Transition Models for Discrete Systems

Discrete event systems encompass a wide variety of physical systems that arise in technology. Typical examples occur in applications such as process control, flexible manufacturing systems, robotics, communication networks, traffic systems, avionics, and embedded real-time computer systems. The processes associated with these

Address correspondence to J. S. Ostroff, Computer Science Department, York University, 4700 Keele Street, North York, Ontario, Canada M3J 1P3.
Received July 1988; accepted August 1989.
*This work is supported by the Natural Sciences and Engineering Research Council of Canada.

655 Avenue of the Americas, New York, NY 10010 0743-1066/91/$3.50

systems may be thought of as discrete in time and space, asynchronous (event-driven rather than clock-driven), and in some sense nondeterministic (capable of "choices" by some mechanism unmodeled by the system analyst). The underlying primitive concepts include events (transitions), states, conditions, and signals. In real-time discrete event processes, system correctness depends not only on the logical result of the system behavior, but also on the time at which the results are produced.

In [28–31], the notion of a *timed transition model* (TTM)[1] was introduced as an extension of the "fair transition systems" defined by Manna and Pnueli [32, 24, 33] —the main extension being the addition of a time metric via upper and lower time bounds on transitions. A TTM (like fair transition systems) is an abstract computational device for representing a variety of models of distributed or parallel computation (e.g. message passing, shared variables, or Petri nets). In addition, a TTM may be used for representing physical devices and processes (e.g. pumps, hardware interlocks, and other mechanical processes), as well as real-time software processes with delay and timeout constructs. Once a suitable TTM is fixed for a given model of computation, it is straightforward to construct a real-time temporal-logic proof theory in which properties of systems expressed in that model can be verified.

The difficulty of modeling and designing real-time discrete event systems has long been recognized in the software engineering literature [41, 34, 38]. As a result, a variety of methods have been proposed for the analysis of real-time systems, including programming languages [15, 39, 16, 14, 21], state machines [42, 1, 7], Petri nets [25, 35, 43, 36, 19, 17], predicate logic and Pressburger arithmetic [12, 26], algebraic approaches [20, 37, 22, 18], and real-time temporal logic [3, 13, 23, 10, 27]. While automated verification techniques for some of these frameworks have been given for finite-state systems (e.g. see [12, 29]), not much work has been done on large (possibly infinite-state) systems. In this paper, the emphasis is on providing mechanized help for the verification of large systems, using the temporal-logic proof system developed in [28].

The following description summarizes the main features of TTMs—the reader is referred to [28, 29] for complete details. A TTM M, for a system composed of discrete event processes, is constructed by enumerating all the system *transitions* together with their order of occurrence. A transition τ consists of a 4-tuple $(e_\tau, h_\tau, l_\tau, u_\tau)$, where e_τ is the enabling condition, h_τ is a state transformation function, and l_τ, u_τ are the lower and upper time bounds. Given that M starts in some initial state s_0, an abstract operational semantics for M is the set of all *trajectories* (infinite sequences of states)

$$s_0 s_1 s_2 s_3 s_4 \cdots s_i s_{i+1} \cdots$$

generated by the TTM. More than one transition may be simultaneously enabled in a given state. Let τ be any transition that is enabled in some state s_i. If τ occurs in s_i, then the successor state s_{i+1} is computed by applying the state transformation function of τ to s_i.

[1] In earlier papers TTMs were called extended state machines. The name was changed to TTMs to emphasized the *timed* features of transition systems.

A conceptual clock is assumed to tick infinitely often during the course of a trajectory. At each tick of the clock the clock variable t is incremented by 1, all other variables remaining the same. The upper-time-bound requirement asserts: if any transition τ becomes enabled in some state of the trajectory, then τ must occur in some subsequent state after no more than (its upper time bound) u_τ ticks of the clock, unless it is preempted by the occurrence of some other transition that disables τ. The lower-time-bound requirement asserts: if any transition τ becomes enabled in some state, it is prevented from occurring in subsequent states for (its lower time bound) l_τ ticks of the clock. Sequences of states that are executed by M subject to the time-bound requirements are possible behaviors of M, and are called trajectories.

The set of all TTMs can be thought of as defining a representation language for modeling and simulating discrete event systems. Such a representation language has a prescriptive, operational nature, and graphic representations (e.g. via transition graphs) can be provided to the designer to help him visualize the structure of each process of the system. In contrast to the prescriptive and operational nature of the representation language, there is also an assertion language for specifying the behavior of M, and for verifying that the behavior is achieved. The assertion language used for TTMs is temporal logic. The verification problem can then be posed as follows: given a TTM M, show that some real-time temporal-logic specification S_M of required behavior is valid for M. S_M is valid if all trajectories generated by M satisfy the specification S_M.

1.2. Constraint Logic Programming

The proofs used in system verification usually involve a tremendous amount of (often trivial) detail that even a disciplined designer will find time-consuming to check. Mistakes are easy to make, and thus the prime purpose of introducing the verification formalism in the first place (i.e. to increase our confidence in the correctness of the system) is threatened. It is impossible to obtain a decision procedure that will automate verification and synthesis of arbitrary systems; nevertheless as much help as possible should be provided to the designer.

The purpose of this paper is to show that the constraint logic-programming language[2] CLP($\Re$) (hereafter called CLP) is a useful computational logic both for representing real-time transition systems and for semiautomating their verification. CLP has an operational model similar to PROLOG. A major difference is that unification is replaced by the more general mechanism of solving constraints in the domain of uninterpreted functors over real arithmetic terms. The following advantages accrue from the use of CLP:

CLP facts and rules represent the designer's knowledge about real-time transition systems in a natural fashion (via arithmetic constraints) that stays close to their mathematical representation. The knowledge is separated from the

[2]For a manual see *The* CLP($\Re$) *Programmer's Manual*, Version 2.0, June 1987, available from the Department of Computer Science, Monash University, Australia.

use that the knowledge is put to, and thus it is relatively easy to effect changes.

The designer is given help in the construction of a class of proofs conveniently presented via proof diagrams, and the correctness of proof diagrams (once constructed) can be checked automatically. The problem of combinatorial explosion of states, encountered in state-by-state checking of proof diagrams, is alleviated by the fact that CLP exploits the rich structure provided by real numbers to efficiently solve real-valued constraints, thus reducing the size of the search space. Thus "large" (even infinite) state spaces can be searched and checked.

There are more powerful theorem provers [5] than CLP, and resolution theorem provers have been suggested for linear-time temporal logic (e.g. [2]). However, what is needed is a computational language that will be suitable both as an (axiomatic) assertion language and for representing the operational nature of TTMs. Constraint logic programming is a suitable compromise: transition systems are easy to represent in a fashion that stays close to their mathematical definition, and many useful assertions for reasoning about the transition system can be posed as constraint-satisfaction problems. Current research into mixed transition systems [4], which include both discrete and continuous variables, require the use of real numbers. The axioms for real numbers need not be asserted in CLP, as CLP has the built-in ability to reason about the uncountable domain of real numbers. There are other interesting constraint logic-programming languages such as CHIP [8], Trilogy [40], and PROLOG-III [6]; however, these other languages do not deal with constraints over real numbers.

The train-gate example used in this paper for illustrating TTMs and proof diagrams is a slightly altered version of the example used in [30]. For simplicity, the train-gate example is somewhat contrived and "small" (in the numbers of processes, variables, and states[3]). However, CLP has been used to verify proof diagrams of larger systems (e.g. see the shared-track example in [28], which has about 10^{15} states), where straight state generation and subsequent testing would be ruled out because of the time complexity involved. The advantage of CLP is that it performs its checks on the state space without the knowledge engineer explicitly adding control information to determine the order of state generation and constraint testing. This alleviates the problem of combinatorial state explosion.

1.3. Organization of the Paper

In Section 2 the notion of a TTM is illustrated with the train-gate example. Section 3 describes the use of proof diagrams for verifying specifications. Section 4 describes how CLP may be used for simulation, proof-diagram construction, and verification. Simulation and verification are illustrated with the train-gate example. Section 5 concludes with a discussion of the use of CLP for automated verification.

[3]Strictly speaking, the state space is infinite because of the addition of the counter variable z. However, the counter plays no part in the verification, and thus can easily be "projected out" of the state space, leaving it finite. An interesting consequence of using CLP is that the elimination of z does not have to be done by the designer (knowledge engineer), but is done automatically by the CLP interpreter. CLP automatically "projects" the answer constraints onto the goal variables.

2. EXAMPLE OF A TTM: A TRAIN SYSTEM

This section describes the train-gate example which will be used to illustrate the notion of a timed transition model. The reader is referred to [29,31] for a formal definition of all terms mentioned below, such as activities, events, activity and data variables, and parallel composition of TTMs.

In the transition graph of Figure 1, a railway crossing consists of three processes: a train, gate, and controller. Each process will be represented by a TTM. The complete train system is also represented by a TTM called *trainGate*, which consists of the parallel composition of the TTMs plant and controller:

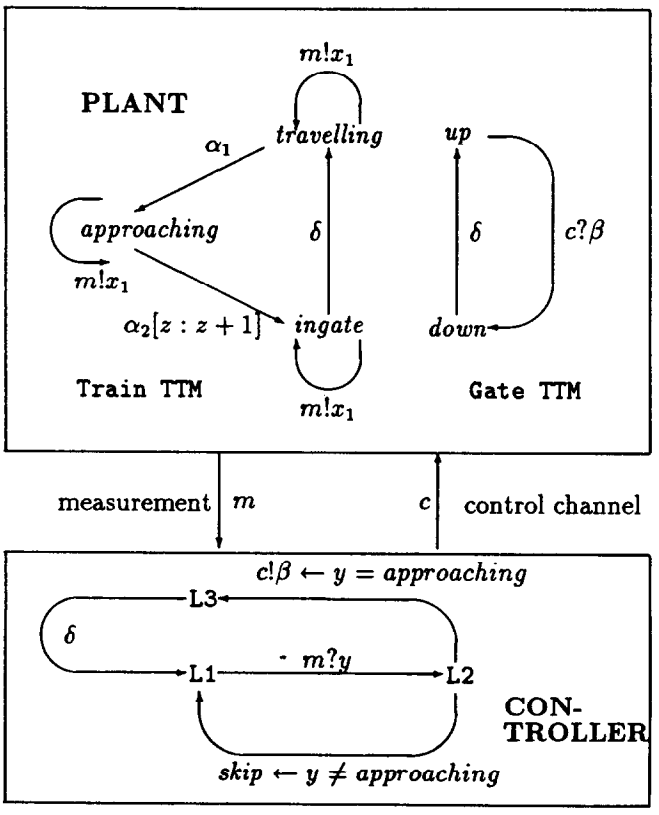
$$trainGate = plant \parallel controller.$$

In turn, the plant consists of two devices:

$$plant = train \parallel gate,$$

where the TTM *train* represents trains that approach the railway crossing (with event label α_1), enter the level crossing (α_2), and then depart (δ), and the TTM *gate* represents a gate that can be lowered (β) or raised (δ). The event label δ is

FIGURE 1. TTM representation of the train-gate system. Train activity variable: $x_1 \in \{traveling, approaching, ingate\}$; train data variable: $z \in \{0, 1, 2, \ldots\}$; gate activity variable: $x_2 \in \{up, down\}$; controller activity variable: $x_3 \in \{\mathsf{L1}, \mathsf{L2}, \mathsf{L3}\}$; controller data variable: $y \in \{traveling, approaching, ingate\}$.



shared by the train and the gate (more on this later). The event α_2 has a lower time bound l_{α_2} which stems from the train inertia; i.e., from the time that the train first approaches the crossing, the clock ticks l_{α_2} ticks before the train reaches the level crossing. The upper time bound is infinity, as there is no guarantee that the train will actually reach the level crossing (it may break down).

The controller consists of a single process represented by the TTM *controller*. In general, a controller may also be constructed from the parallel composition of many TTMs. Controller TTMs are usually implemented as software processes in a real-time programming language, so that changes can easily be made [28]. Programming constructs such as delays and timeouts can be modeled by lower and upper time bounds on the corresponding TTM transitions.

The plant represents that part of the system that is given and whose structure cannot be altered. However, there is usually the possibility of influencing the behavior of the plant by appropriate control actions. For the plant of the train-gate example, measurements of the train state can be made over the channel m, and based on the current measurements, a decision can be made by the controller to issue a command over the channel c for the gate to be lowered. The gate edge labeled $c?\beta$ means the gate is waiting to receive a command from the controller over the channel c to lower the gate.[4]

In contrast to the plant, the controller is that part of the overall system *trainGate* that can be altered by the designer to achieve his goals. In the case of *trainGate*, the controller has been designed so as to prevent the gate from being up while simultaneously the train is on the level crossing. Also, the gate should not be lowered unnecessarily (i.e., once the gate has been raised, it should not be lowered unless a train once again approaches).

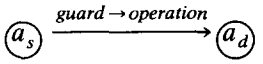
Each process has a distinguished variable called the *activity variable*. For example, the activity variable of the TTM *train* is x_1, with associated range $type(x_1) = \{traveling, approaching, ingate\}$. Each element in $type(x_1)$ is called an *activity* of the TTM *train*. Thus the activities of the train are *traveling*, *approaching* (close to the crossing), and *ingate* (actually on the crossing). The *activities* of the gate are *up* and *down*. Events (transitions) such as β (the gate is lowered) and α_1 (the train enters the approach zone) are assumed to occur instantaneously, while activities have duration in time.

In addition to the activity variables of TTMs (for control information), each TTM can optionally have a set of *data variables*. Data variables represent numerical information (such as temperatures, pressures, levels, or counters) in the plant, and can also be used for program variables in software processes. For example, the train has a counter z [where $type(z)$ is the set of natural numbers] for counting the number of trains that have crossed over the level crossing. If the transition α_2 occurs in a state s of *trainGate*, then the successor state s' coincides with s, except that the value of x_1 is *ingate*, and the value of z is $s(z)+1$ [where $s(z)$ is the value of z in the state s—i.e., the counter z is incremented by 1 on the occurrence of α_2]. The transformation function h_{α_2} of α_2 is denoted $[x_1 : in, z : z+1]$ (so that only those components that are changed are indicated).

[4]The controller operation $c!\beta$ means "send the command β on channel c to the gate". The notations $c!\beta$ and $c?\beta$ are borrowed from synchronous communicating actions in CSP [9].

The controller process has a data variable y, which is used to store the current measurement of train activity, and thus has the same type as x_1.

An edge such as



in a transition-graph representation of a TTM has the intepretation: "if the TTM is currently in activity a_s and if the *guard* evaluates to *true*, then the edge is traversed while doing *operation*, after which the TTM is in activity a_d". If the guard is left out, then it is assumed to be *true*. The *operation* may be an assignment, or a communication involving the sending or receiving of a message. It is a straightforward matter to translate an edge of a transition graph into a transition. The enabling condition is $(guard \wedge x = a_s)$, where x is the activity variable of the process containing the edge. If *operation* is the simultaneous assignment $v_1 : a_1, \ldots, v_n : a_n$ of data variables v_1 to v_n to expressions a_1 to a_n, then the transformation function is $[x : a_d, operation]$. A similar translation procedure can be given for linear text programs, Petri nets, or communicating sequential processes [33].

Operations can be used to communicate or to update data variables. There are three kinds of operations:

1. An example of an *assignment* operation is $\alpha_2[z : z + 1]$, which is read "the value of the expression $z + 1$ is assigned to z". Multiple simultaneous assignments (e.g. 2) are denoted $\alpha[y_1 : a_1, y_2 : a_2]$.
2. An example of a *send* operation is $m!x_1$, meaning that the current value of the activity variable x_1 is sent on channel m.
3. An example of a receive operation is $m?y$, meaning that the message received on channel m is placed in the data variable y.

As in CSP [9], we arrange for TTMs to synchronize their events when they need to interact. In this way, the lower-level handshaking (e.g., via semiphores, monitors, or condition queues) may be ignored. The synchronized action consisting of the simultaneous participation of component events is called an *interaction*. For example, the interaction consisting of the edges with operation $m!x_1$ in the train and $m?y$ in the controller forms a *communicating transition* (called *m-chan*), in which there is a distributed assignment of the value of x_1 to y. As another example of an interaction, all three TTMs share an event label δ. For the train TTM, δ represents the train exiting from the crossings, for the gate TTM it represents the raising of the gate, and in the controller TTM it represents a "reset" or controller initialize. The three edges together form a *shared transition*. The resulting shared transition involves the simultaneous action in which the train exits, the gate is raised, and the controller is reset.

The total behavior of *trainGate* can be described by the six transitions listed in Table 1. Three of the transitions are interactions and the other three are the "local" actions α_1, α_2, and *skip*. Two of the interactions are communication transitions (one over the channel c and the other over the channel m), and the third interaction δ is a shared transition. In the table, *tr* is an abbreviation for *travelling*, *in* for *ingate*, and *ap* for *approaching*.

TABLE 1. Transitions for the train-gate example

Name	Enabling condition	Transformation	Lower	Upper
α_1	$(x_1 = tr)$	$[x_1 : ap]$	0	∞
α_2	$(x_1 = ap)$	$[x_1 : in, z : z + 1]$	10	∞
β	$(x_2 = up \wedge x_3 = \mathsf{L2} \wedge y = ap)$	$[x_2 : down, x_3 : L3]$	0	u_β
δ	$(x_1 = in \wedge x_2 = down \wedge x_3 = \mathsf{L3})$	$[x_1 : tr, x_2 : up, x_3 : \mathsf{L1}]$	0	∞
m-chan	$(x_3 = \mathsf{L1})$	$[x_3 : \mathsf{L}_2, y : x_1]$	0	u_m
skip	$(x_3 = \mathsf{L2} \wedge y \neq ap)$	$[x_3 : \mathsf{L1}]$	0	u_s

Each transition τ in Table 1 consists of an enabling condition e_τ, a transformation function h_τ, and lower and upper time bounds l_τ and u_τ. The set of all transitions of *trainGate* is denoted $\mathscr{T}_{trainGate}$, and $\mathscr{V}_{trainGate}$ denotes the set of variables of the TTM *trainGate* (including activity variables such as x_1 and data variables such as z).

Usually there is more than one initial state for a given system. The initial condition $\Theta_{trainGate}$ is a predicate whose satisfying states are exactly the initial states of *trainGate*. For example, the initial condition

$$\Theta_{trainGate} \stackrel{\text{def}}{=} (x_1 = travelling \wedge x_2 = up \wedge x_3 = \mathsf{L1})$$

specifies all states in which the train is not yet near the crossing, the gate is up, and the controller is at L1. Neither the train counter variable z nor the controller data variable y is initially restricted.

A precise definition of a TTM is given by $M = (\mathscr{V}, \Theta, \mathscr{T})$,where $\mathscr{V}$ is the set of all TTM variables, Θ its initial condition, and $\mathscr{T}$ its set of transitions. A state s is a mapping from the variables set to the values (or types) of the variables. The abstract operational semantics of the TTM is given by all its possible trajectories (or sequences of states) as mentioned in the Introduction.

The TTM *trainGate* is thus precisely defined as

$$trainGate \stackrel{\text{def}}{=} (\mathscr{V}_{trainGate}, \Theta_{trainGate}, \mathscr{T}_{trainGate}).$$

The CLP representation of *trainGate* follows very closely to its mathematical definition as given above (see Section 4).

3. TEMPORAL LOGIC AND PROOF DIAGRAMS

This section illustrates the use of temporal logic for specifying real-time discrete event systems, and summarizes the construction of proof diagrams and their use in verification. This summary provides the basis for the subsequent discussion of how CLP automates verification. For a more detailed discussion of real-time temporal logic and proof diagrams and an account of the sound proof system used in this section, the reader is referred to [28].

Real-time temporal logic allows the expression of a variety of TTM properties, e.g.

safety properties such as $\Box P$—henceforth the property P will be true,

liveness properties such as $\Diamond P$—the property P must eventually become true, and

real-time response properties such as $\Diamond(P \wedge t = 6)$—the property P must eventually become true at time 6 ticks (of the clock variable t).

A temporal specification S of a TTM is *satisfied* in a trajectory of the TTM if S evaluates to *true* in the trajectory. For example, $\Diamond P$ is satisfied in the trajectory $s_0 s_1 s_2 \ldots s_i \ldots$ if there is some state s_i in which the property P holds true.

For simplicity, consider the *invariance*[5] property for the train-gate system specified as

$$S_{trainGate} \stackrel{\text{def}}{=} \Box \neg (x_1 = ingate \wedge x_2 = up),$$

meaning "*henceforth* it is not the case that the gate is up while simultaneously the train is crossing." $S_{traingate}$ is a *valid property of M* if all possible trajectories executed by *trainGate* satisfy $S_{trainGate}$.

The following notation is used in the sequel:

$\varphi, \varphi_0, \varphi_1, \ldots$ and $\psi, \psi_1, \ldots$ denote *state formulas*. State formulas are predicates that can be evaluated to *true* or *false* in a single state. By contrast, temporal formulas containing temporal operators can only be evaluated in trajectories (sequence of states). The initial condition $\Theta_{trainGate}$ is an example of a state formula. A state formula is *valid* if it evaluates to true in all states. Given a transition τ with transformation functions $[v_1 : a_1, \ldots, v_k : a_k]$, we let φ^τ denote $\varphi^{v_1, \ldots, v_k}_{a_1, \ldots, a_k}$ (i.e., all free occurrences of variables $v_1, \ldots, v_k$ in φ are simultaneously replaced by the expressions $a_1, \ldots, a_k$ respectively).

For any transition τ, if $(e_\tau \wedge \varphi_1) \rightarrow \varphi_2^\tau$ is satisfied in every state, then we say "τ leads from φ_1 to φ_2", and we write $\{\varphi_1\}\tau\{\varphi_2\}$. The condition $(e_\tau \wedge \varphi_1) \rightarrow \varphi_2^\tau$ is similar to the Hoare logic axiom for the assignment statement, except that the enabling condition is used to strengthen the precondition. As will be seen, an important feature of CLP is that it performs an efficient check for the validity of state formulas such as $(e_\tau \wedge \varphi_1) \rightarrow \varphi_2^\tau$.

For any set of transitions $\mathcal{T}$, $\{\varphi_1\}\mathcal{T}\{\varphi_2\}$ is an abbreviation for: $\{\varphi_1\}\tau\{\varphi_2\}$ for each $\tau \in \mathcal{T}$.

[5]The rest of this paper will focus on the use of CLP for constructing and checking proof diagrams for invariances. However, the same notions can be used for checking *eventualities* and other temporal properties. For eventualities, in addition to checks similar to those used for invariances, two other checks are needed: the diagram must be acyclic, and certain "progress" transitions must be enabled in all states of a node [28]. These additional checks are easily implemented in CLP.

A typical proof rule for proving invariances of real-time temporal logic is the rule INV given by

$$\frac{\{\psi\}\,\mathscr{T}\,\{\psi\} \qquad \Theta \rightarrow \psi}{\Box\,\psi}$$

The above proof rule may be illustrated with a proof diagram, i.e. a "high-level" version of state reachability graph. To prove $S_{trainGate}$ using INV, it is sufficient to perform the following checks:

1. Start with a state formula ψ_0 (called an initial node of the proof diagram) that is either the initial condition $\Theta_{trainGate}$ or a weakened version thereof, and check the validity of

$$\psi_0 \rightarrow \neg(x_1 = ingate \wedge x_2 = up) \tag{1}$$

as required by the second hypothesis of INV. In addition:

2. Systematically explore and check all possible developments to new nodes from the initial node. Let φ be the disjunction of all nodes in the proof diagram. It trivially follows that $\psi_0 \rightarrow \varphi$ is valid, and since $\Theta_{trainGate} \rightarrow \psi_0$, the validity of

$$\Theta_{trainGate} \rightarrow \varphi \tag{2}$$

follows. Every transition that is enabled in some state satisfying the initial node may lead to a new node, which in turn must be systematically checked for exiting transitions. If for each node ψ_i in the proof diagram

$$\psi_i \rightarrow \neg(x_1 = ingate \wedge x_2 = up) \tag{3}$$

(i.e. all states satisfying the node also satisfy the required property) and

$$\{\psi_i\}\,\mathscr{T}_{trainGate}\{\varphi\} \tag{4}$$

(i.e. all transitions that exit from a node lead to another node) are valid, then the validity of

$$\{\varphi\}\,\mathscr{T}_{trainGate}\{\varphi\} \tag{5}$$

trivially follows. From the formulas (5), (2) and INV we obtain $\Box\,\varphi$. Since by (3) it follows that $\varphi \rightarrow \neg(x_1 = ingate \wedge x_2 = up)$ is valid, it then trivially follows from temporal reasoning that $S_{trainGate}$ is also valid.

Since each node (representing a state formula) in the proof diagram has a possibly infinite set of satisfying states, it is possible to obtain a finite number of nodes and edges to be checked, even if there are an infinite number of states.

The proof of specification $S_{trainGate}$ is illustrated in the proof diagram of Fig. 2. An arrow labeled τ from some node ψ_i to ψ_j means that $\{\psi_i\}\,\tau\,\{\psi_j\}$ is valid.

In Figure 2, the transition α_2 does not always lead to a suitable node, which is why no node is shown at the arrow destination in some cases. In fact, at the three nodes ψ_1, ψ_2, and ψ_3, for transition α_2, the formula given in (3) is not valid, because α_2 leads to a state formula that is inconsistent with $\neg(x1 = ingate \wedge x_2 = up)$. The INV proof rule on its own is not strong enough to prove the required

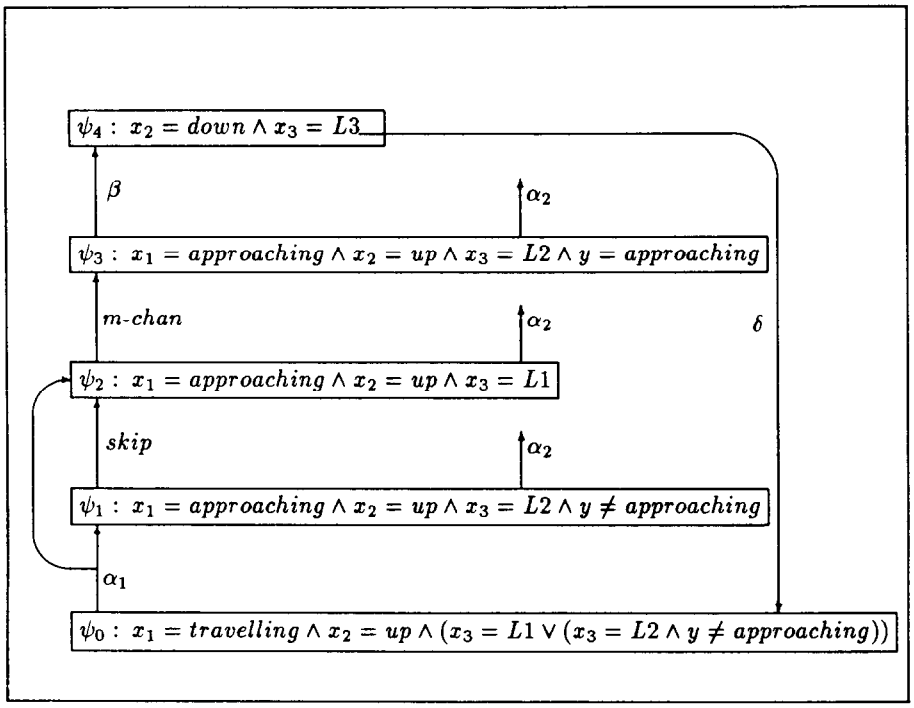


FIGURE 2. Proof diagram for train-gate example.

property, because it does not use the information conveyed by the time bounds. It is by means of the time bounds that it can be shown that α_2 will not occur from the three nodes, and hence that the required invariance is preserved. Thus, although the proof rule indicates to us where real-time reasoning must be employed (in this case with respect to α_2), the proof rule on its own is not sufficient to verify the required safety property. What additional conditions must INV be supplemented with to prove the required property?

We must show that α_2 will never occur from the nodes ψ_1, ψ_2, and ψ_3, i.e., we must demonstrate that the sequence of three transitions *skip* .. *m-chan* .. β always occurs before α_2 can occur; this can be argued for, so long as [28]

1. the inequality

$$u_{skip} + u_{m\text{-}chan} + u_\beta < l_{\alpha_2} \tag{6}$$

holds true, and

2. *skip*, *m-chan*, and β are enabled in any state satisfying the nodes ψ_1, ψ_2, and ψ_3 respectively. A transition τ is enabled in all states of a node ψ_i if

$$\psi_i \rightarrow e_\tau \tag{7}$$

is valid.

According to (4), each transition in $\mathscr{T}_{trainGate}$ must be checked to see that it leads from a node to some other node. In fact, it is necessary to check only the exiting transitions. The following will define the notion of an existing transition. Let s be any state satisfying a node ψ_i, and let τ be any transition. There are three possibilities. Either

τ **is disabled in** ψ_i—i.e., $\psi_i \wedge e_\tau$ evaluates to *false* in s $[s(\psi_i \wedge e_\tau) = false]$, or

τ **is enabled in** ψ_i—i.e.,

$$s(\psi_i \wedge e_\tau) = true \tag{8}$$

in which case τ has a successor state s', and

either τ **is a self-loop in** ψ_i—i.e. $s(\psi_i \wedge e_\tau \wedge \psi_i^\tau) = true$. If the transition τ occurs from s it leads back to the node ψ_i, i.e. s' satisfies ψ_i.

or τ **exits from** ψ_i—i.e., $s(\psi_i \wedge e_\tau \wedge \neg \psi_i^\tau) = true$. After the occurrence of τ, the successor state s' does not satisfy ψ_i. If the successor state s' satisfies another node ψ_j in the proof diagram, then $s(\psi_i \wedge e_\tau \wedge \psi_j^\tau) = true$; in such a case we say that there is a path from ψ_i to ψ_j via τ.[6] Note that the following equivalence is valid:

$$[\psi_i \wedge e_\tau \wedge \neg \psi_i^\tau] \leftrightarrow [\neg(\{\psi_i\}\, \tau\, \{\psi_i\})]. \tag{9}$$

If a transition τ either is disabled in all states satisfying a node ψ_i, or is a self-loop in all states satisfying a node ψ_i, then the validity of $\{\psi_i\}\tau\{\varphi\}$ trivially follows. Thus when checking the formula (4), it is sufficient to check only the exiting transitions. For example, in the case of the initial node ψ_0, the only exiting transition is α_1. In fact, α_1 exists in all states for which it is enabled. The successor node can be found automatically by computing the strongest postcondition of α_1 with respect to ψ_0 (see [28] for the details).

In Figure 2, each edge indicates an exiting transition. An edge labeled τ should only be drawn from a source node φ_s to a set of destination nodes $\varphi_1, \ldots, \varphi_k$ if $\{\varphi_s\}\tau\{\varphi_1 \vee \cdots \vee \varphi_k\}$ holds true. As an example, the forked arrow labeled α_1 indicates that $\{\psi_0\}\,\alpha_1\,\{\psi_1 \vee \psi_2\}$ is true. Self-loops are not shown in the diagram. In other words, the proof diagram only shows those transitions that are enabled and lead to a destination node that is different from the source node.

Given an arbitrary specification such as $S_{trainGate}$, CLP is used in the sequel both to aid in the construction of a suitable proof diagram and to automatically verify a given proof diagram once constructed. Based on the discussion given in this section, we summarize below the key points involved in constructing and verifying proof diagrams, before proceeding to the next section for the CLP discussion.

The heuristic for constructing a proof diagram is:

1. Start from an initial node ψ_0 and check the validity of (1).
2. Generate new nodes by exploring all exiting transitions from the current nodes. Compute destination nodes using strongest postconditions. For each new node check the validity of (3). If (3) fails to hold for a particular transition, try to satisfy the timing conditions (6) and (7).

[6]Note that τ "leads" from ψ_0 to φ iff for *any* state s, $s(\psi_i \wedge e_\tau \rightarrow (\psi_j)^\tau) = true$. Thus the "leads to" and "path" relations should not be confused.

To verify the property $S_{trainGate}$ given a proof diagram such as Figure 2, it is sufficient to establish the following [28]:

1. For each node ψ_i such that $0 \leq i \leq 4$, show the validity of (3).
2. For each node ψ_i show that each exiting transition τ (except for the transition α_2) leads to $\varphi \stackrel{\text{def}}{=} (\psi_0 \vee \cdots \vee \psi_4)$, i.e., show the validity of (4).
3. To take care of α_2 where it exits without a suitable destination node, check the validity of (7) and verify the inequality (6).

The main detail in the above proof of $S_{trainGate}$ involves propositional and predicate (and in larger examples arithmetic) reasoning. The temporal-logic component of the reasoning is quite small. It is for the nontemporal part of the reasoning that CLP is best employed. The check of step 2 above is time-consuming, as each transition must be checked to see if it exits from each node in the proof diagram. Although the sequel focuses mainly on step 2, each of the other steps can also be checked automatically using CLP.

A final point, which will not be explored any further in this paper, is that the real-time part of the reasoning often requires the simultaneous solution of inequalities such as (6). CLP is obviously well suited for solving such scheduling inequalities, where some of the upper or lower time bounds are unknown.

4. CONSTRAINT LOGIC PROGRAMMING FOR TTM'S

Two CLP components are involved in representing and reasoning about TTMs: a knowledge base for representing the particular TTM that is to be analyzed, and a generic query model that is used to analyze a given knowledge base.

Knowledge base. A TTM M is represented by a knowledge base of CLP facts, rules, and arithmetic constraints, in a way that stays close to the mathematical representation of transition systems. The knowledge base for M is split into two components: one component resides in the file `M.ttm`, and the other in `M.sf`. The names `M.ttm`, `M.sf` will be used interchangeably both for the CLP code and the corresponding file names.

`M.ttm` is the CLP description of M, including the types of its variables, its enabling conditions, its transformation functions, its initial condition, and (optionally) its lower and upper time bounds.

`M.sf` is the CLP component that represents all the nodes in a proof diagram corresponding to a given specification of M. Each node (state formula) in the proof diagram is represented by an arithmetic constraint.

Query Module. The query module `QM` uses CLP as the inference engine to query the knowledge base with questions in areas such as:

Simple simulation. Given the current state of M, what transitions are enabled, and what are the successor states?

Proof diagrams. What transitions are enabled in a given node of a proof diagram, what new nodes do exiting transitions lead to, and is a given proof diagram correct? `QM` semiautomates the construction of proof dia-

grams, and automates the check that a given proof diagram is correct. Even though the knowledge base will obviously change depending on which TTM is examined, the same query module `QM` is always used.

Detailed knowledge of CLP is not necessary for the correct understanding and use of `QM`, though an understanding of the PROLOG inference mechanism (from which CLP is derived) is useful.

Before discussing knowledge bases and the query module in detail, a brief discussion of the features of CLP is in order.

4.1. Features of CLP

The ability of CLP to deal with constraints directly (rather than with the set of states represented by the constraints) significantly eases the problem of combinatorial explosion of states. For example, let `X` represent a TTM variable which ranges over the reals, and let some transition have enabling condition `X > 1`. Suppose a node in the proof diagram corresponds to the state formula `X < 1`. The transition is enabled in some state of the node if there is at least one state satisfying the conjunction of the two constraints [see (8)]; the appropriate query is thus

```
| ?- X < 1, X > 1.
```

to which the expected answer is `*** No ***`, i.e. there is no state of the node from which the transition is enabled.

If the above query is submitted to standard (non-CLP) implementations of PROLOG, an arithmetic-expression error results, as `X` is uninstantiated; the declarative bidirectional nature of PROLOG does not extend to arithmetic expressions. Standard PROLOG could be used to check the above conjunction with a generate (the states) and test (for satisfaction) method, but such a method is obviously unsatisfactory for checking large state spaces.

In contrast to PROLOG, submission of the above query to CLP immediately yields the expected answer `*** No ***`. The current implementation of CLP uses Gaussian elimination to solve linear inequalities, and the first phase of the simplex algorithm to check the solvability of linear inequalities. Thus, by using CLP, an explicit state-by-state check (impossible anyway if there are an infinite number of states) is replaced by direct symbolic manipulation of the constraints. Thus, CLP is useful in proof-diagram construction because a question concerning satisfaction of state formulas in a state can be transformed into a question concerning the existence of solutions to a set of constraints.

Constraints. Constraints in CLP consist of equations, inequalities, and inequations built up from the application of the following operators to real constants and variables:

1. the functions `+, -, *, /`,
2. the relations `=, >=, >, <=, <`, and
3. the relation `neq`, where `X neq Y` denote the inequation $X \neq Y$.

The symbols in items 1 and 2 have their usual meaning, and parentheses may be used to resolve ambiguities. The `neq` relation is not in the current implementation

of CLP. However, the algorithm used for equations may also be applied to inequations, and thus future implementations of CLP may be expected to deal with them. Appendix A contains a definition of `neq` that suffices for the present but is somewhat inefficient. The CLP manual may also be consulted for the description of many other special functors (e.g., `sin`). The inequality symbol is implemented in PROLOG-III.

4.2. *Building the Knowledge Base*

The train-gate example (see Figure 1) will be used to illustrate how CLP represents TTMs.

4.2.1. `trainGate.ttm`—*Representing TTMs.* Before dealing with the enabling conditions and transformation functions of the TTM, *trainGate* we must first define the type of each TTM variable. The CLP fragment below defines the variable types, as well as a special predicate `map`.

```
%       Var     Type          range
%       ---     ----          -----
%       X1      tr,ap,in       0,1,2
%       X2      up,down        0,1
%       X3      L1,L2,L3       0,1,2
%       Y       tr,ap,in       0,1,2

type(trainGate,x1,X1):-X1>=0, X1<=2.
type(trainGate,x2,X2):-X2>=0, X2<=1.
type(trainGate,x3,X3):-X3>=0, X3<=2.
type(trainGate,y,Y):-Y>=0, Y<=2.
type(trainGate,z,Z):-Z>=0.

% map/3 is read: ``The state-vector [X1,X2,X3,Y,Z] is mapped to values in
                   the state-space of trainGate''.
% map(trainGate, [x1,x2,x3,y,z],[X1,X2,X3,Y,Z]):-
        type(trainGate,x1,X1),
        type(trainGate,x2,X2),
        type(trainGate,x3,X3),
        type(trainGate,y,Y),
        type(trainGate,z,Z).
%map/2 is a short way of calling map/3 and of obtaining the order of
%the TTM variables.
map(Ttm,Statevector):-map(Ttm,_,Statevector).
```

To use the arithmetic constraint mechanism of CLP, activities must be transformed into sequentially ordered sets of integers. Thus the activity *traveling* is 0, *approaching* is 1, and *ingate* is 2. Clearly, there is no problem representing the types of variables such as z which have infinite ranges.

A CLP variable is a logical variable defined within the scope of a rule or query, but having no meaning outside of its scope. By contrast, a TTM variable is "global" and thus should have the same meaning anywhere in the knowledge base.

If a CLP variable (e.g. X1) is to represent a TTM variable (e.g. the train activity variable x_1), then there must be some way in which X1 can be "mapped" onto the *type* of x_1. The purpose of the map predicate is to constrain all the CLP variables representing TTM variables to the appropriate type (via the type declaration). Thus, map must be used at the beginning of any query to the knowledge base, as will be illustrated in the sequel.

ENABLING CONDITIONS. Enabling conditions for *trainGate* (from Table 1) are written as follows:

```
en(trainGate,alpha1,[X1,X2,X3,Y,Z]):-
        X1=0.
en(trainGate,alpha2,[X1,X2,X3,Y,Z]):-
        X1=1.
en(trainGate,beta,[X1,X2,X3,Y,Z]):-
        X2=0,X3=1,Y=1.
en(trainGate,delta,[X1,X2,X3,Y,Z]):-
        X1=2,X2=1,X3=2.
en(trainGate,m_chan,[X1,X2,X3,Y,Z]):-
        X3=0
en(trainGate,skip,[X1,X2,X3,Y,Z]):-
        X3=1,Y neq 1.
```

For example, en(trainGate,alpha1,[X1,X2,X3,Y,Z]) is read declaratively as "the *trainGate* transition alpha1 is enabled in any state satisfying $x_1 = 0$". To produce all transitions that are enabled in the state

$$\{(x_1, approaching), (x_2, up), (x_3, L1), (y, approaching), (z, 21)\} \tag{10}$$

the following query can be used:

```
1 ?- map(trainGate,[X1,X2,X3,Y,Z]),
      en(trainGate,L,[X1,X2,X3,Y,Z]),
      X1=1,X2=0,X3=0,Y=1,Z=21.
L=alpha2
X1=1
X2=0
X3=0
Y=1
Z=21
*** Retry *** ? y
L=m_chan
X1=1
X2=0
X3=0
Y=1
Z=21
*** Retry *** ? y
*** No more answers ***
```

The reply indicates that the transitions labelled `alpha2` and `m_chan` are enabled in the given state.

TRANSFORMATION FUNCTIONS. The transformation functions are declared as follows:

```
h(trainGate,alpha1,[X1,X2,X3,Y,Z],[1,X2,X3,Y,Z]).
h(trainGate,alpha2,[X1,X2,X3,Y,Z],[2,X2,X3,Y,Z+1]).
h(trainGate,beta,[X1,X2,X3,Y,Z],[X1,1,2,Y,Z]).
h(trainGate,delta,[X1,X2,X3,Y,Z],[0,0,0,Y,Z]).
h(trainGate,m_chan,[X1,X2,X3,Y,Z],[X1,X2,1,X1,Z]).
h(trainGate,skip,[X1,X2,X3,Y,Z],[X1,X2,0,Y,Z]).
```

An advantage of using CLP can be seen with the representation of the `alpha2` transition involving the counter variable `Z`—the expression `Z + 1` would not be allowed in standard PROLOGs except in queries in which `Z` is instantiated. Any valid arithmetic expression can be used instead of `Z + 1`. The above CLP definition of transformation functions also allow for the computation of simultaneous substitutions such as ψ^{α_2} (see the description of `sfexits` and `path` in Section 4.3).

In a simulation of TTM behavior there is often a need to compute the successor to the current state. The following query obtains successor state `NewQ` to `Q` for the transition α_2, where `Q=[1,0,0,1,21]` corresponding to the state given in (10):

```
2 ?- map(trainGate,Q),h(trainGate,alpha2,Q,NewQ),
     Q=[1,0,0,1,21].
Q=[1,0,0,1,21]
NewQ=[2,0,0,1,22]
*** Retry *** ? y
*** No more answers ***
```

In general, before applying the transformation function to compute a successor state, there should first be a check that the transition is enabled:

```
3 ?- map(trainGate,Q),Q=[X1,X2,X3,Y,Z],
     en(trainGate,alpha1,Q),
     h(trainGate,alpha1,Q,NewQ).
Q=[0,X2,X3,Y,Z]
NewQ=[1,X2,X3,Y,Z]
```

The reply tells us that α_1 is enabled in the state `[0,X2,X3,Y,Z]`, i.e., $x_1 =$ *traveling* and the other components can be anything, and after the occurrence of α_1 the successor state is `[1,X2,X3,Y,Z]`, i.e., $x_1 =$ *approaching* and the other components are unchanged. The above query nicely indicates symbolic execution, as the components in the state vector need not always be instantiated—an answer such as `Q=[0,X2,X3,Y,Z]` in fact represents an infinite number of states.

4.2.2. `trainGate.sf`*—Representing Proof Diagrams.* Nodes in the proof diagram of Figure 2 are represented as follows:

```
sf(trainGate,0,[X1,X2,X3,Y,Z]):-
        X1=0,X2=0,(X3=0; X3=1,Y neq 1).
sf(trainGate,1,[X1,X2,X3,Y,Z]):-
        X1=1,X2=0,X3=1,Y neq 1.
sf(trainGate,2,[X1,X2,X3,Y,Z]):-
        X1=1,X2=0,X3=0.
sf(trainGate,3,[X1,X2,X3,Y,Z]):-
        X1=1,X2=0,X3=1,Y=1.
sf(trainGate,4,[X1,X2,X3,Y,Z]):-
        X2=1,X3=2.
```

The first `sf` predicate above asserts: the node ψ_0 in the proof diagram of *trainGate* is satisfied by a state whose component variables x_1, x_2, x_3, y, and z take on values satisfying the state formula

$$x_1 = \textit{traveling} \wedge x_2 = \textit{up} \wedge (x_3 = \texttt{L1} \vee (x_3 = \texttt{L2} \wedge y \neq \textit{approaching})).$$

Some queries in the sequel require negated versions of the above state formulas. Since negation in CLP (as in PROLOG) is "unsafe", a simple way of getting around this problem is provided in Appendix B.

4.3. `QM`*—The Query Model*

Simple simulation and symbolic execution using the knowledge base was illustrated in Section 4.2. We now illustrate construction and analysis of proof diagrams for verification. Each query supported by `QM` is illustrated with an example—however, the reader should refer to Appendix A for complete details of the CLP code for `QM`.

4.3.1. Determining Enabled and Exiting Transitions. The `sfenabled` predicate may be used to determine which transitions are enabled from a given state formula. For example, to construct the proof diagram in Figure 2 starting from the initial node ψ_0, it must be determined which transitions are enabled from ψ_0, so that successor nodes can be explored. Consider the query:

```
4 ?- sfenabled(trainGate,Transition,0).
Transition = alpha1
*** Retry *** ? y
Transition = m_chan
*** Retry *** ? y
Transition = alpha1
*** Retry *** ? y
Transition = skip
*** Retry *** ? y
Transition = alpha1
```

```
*** Retry *** ? y
Transition = skip
*** Retry *** ? y
*** No more answers ***
```

The answer to the query is that each of α_1, *m-chan*, and *skip* is enabled from ψ_0 (and no other train-gate transitions are so enabled). As indicated by (8), the query asks: "For what train-gate transitions τ, is there a state satisfying the state-formula $(\psi_0 \wedge e_\tau)$?".

Queries such as `sfenabled`, having the state vector `[X1,X2,X3,Y,Z]` as an argument, have two versions: e.g. `sfenabled/4` and `sfenabled/5`. The `sfenabled/5` predicate performs all the tasks of `sfenabled/4`. In addition, it returns the constraints under which the answer is satisfied for reasons explained in Appendix A.

The existence of a disjunction (the semicolon in CLP) in a state formula creates a choice point, so that an alternative derivation is pursued on backtracking. As a result, the same answer `alpha1` (for example) is returned three times. It is straightforward to construct "filters" so that duplicated results are not repeated.

The transitions *m-chan* and *skip* are self-loops, although the `sfenabled` predicate does not indicate this fact. In the construction of proof diagrams for invariances, self-loops need not be explored any further in the proof diagram, as they do not lead to new nodes. The `sfexits` and `exits` predicates ignore self-loops and produce as answers only those transitions that exit from a given node (`sfexits`) or that exit from the proof diagram (`exits`).

The predicate `sfexits` is similar to `sfenabled` except that it filters out self-loops. If there is any state satisfying $(\psi_0 \wedge e_\tau \wedge \neg \psi_0^\tau)$, then, as determined by (9), the transition τ exits from ψ_0. Therefore, a `Yes` answer to the `sfexits` query returns all the transitions that are not self-loops (but are exiting transitions):

```
5 ?- sfexits(trainGate,Transition,0).
Transition = alpha1
```

Thus α_1 is the only transition to exit from the node ψ_0. Construction of the proof diagram can then proceed by exploring the α_1 path, as explained in the summary at the end of Section 3.

The predicate `exits` may be used to check the validity of (4), which is the crucial step in verification. For example, the query

```
8 ?- exits(trainGate,Transition,StateformulaNo).
Transition = alpha2
StateformulaNo = 1
*** Retry *** ?
```

indicates that α_2 exits from ψ_1 to a state not satisfied by any node in the proof diagram. Subsequent retries confirm that α_2 also leads to states inconsistent with the specification at nodes ψ_2 and ψ_3. Thus all paths satisfy the specification $S_{trainGate}$ with the exception of α_2 from ψ_1, ψ_2, ψ_3.

The `exits` predicate checks that for each transition τ and node ψ_i there is a constraint (set of states) satisfying[7] $\neg[\{\psi_i\}\tau\{\psi_0 \vee \cdots \vee \psi_4\}]$. A `No` answer means that there are no such satisfying states, i.e., (4) is valid. A `Yes` answer lists those paths that are unsatisfactory.

4.4. *Verifying Specifications with Proof Diagrams*

The correctness of the proof diagram for `trainGate.sf` can be checked as follows:

1. Set up the knowledge base `trainGate.ttm` (to represent the train gate) and `trainGate.sf` (to store the nodes of the proof diagram[8]).
2. Append to `trainGate.sf` the negated nodes as indicated in Appendix B.
3. Consult `QM`, `trainGate.ttm`, and `trainGate.sf` with CLP.
4. Use `exits` to check[9] the validity of (4). For paths that exit the proof diagram, the checks (6), (7) must be made.

4.5. *Constructing Proof Diagrams*

As mentioned previously, the `sfexits` query determines exiting transitions in the construction of proof diagrams. When only part of the proof diagram has been constructed, and a new node is added, it is necessary to determine if any of the exiting transitions are on paths from the new node to the current one. The predicate `path(Ttm,SFa,Transition,SFb)` determines if there is a path from state formula `SFa` to `SFb` via `Transition` for the stated TTM. If `SFa`, `Transition`, `SFb` are uninstantiated in a query, then `path` will find all paths including self-loops:[10]

```
6 ?- path(trainGate,3,Transition,4).
Transition = beta
*** Retry *** ? y
*** No more answers ***
7 ?- path(trainGate,4,Transition,3).
*** No ***
```

Thus β is the only transition on a path from ψ_3 to ψ_4, and there is no transition on a path from ψ_4 to ψ_3.

[7] See `metanegsf` in Appendix B.

[8] The nodes are assumed to satisfy the formulas specified by (3). CLP may be used to do this check —a `No` answer to the query `sf(trainGate,Nodeno,[X1,X2,X3,Y,Z]), X1=ingate,X2=up` means that (3) is valid.

[9] The complete proof diagram of all five nodes given by $\varphi \stackrel{\text{def}}{=} (\psi_0 \vee \cdots \vee \psi_4)$ is checked by the `exits` query. However, it is not necessary to always check all the nodes simultaneously. A more efficient check is described in [28].

[10] Though it is possible to check complete proof diagrams with `path`, a friendlier version (called `filterpath`) will be described below.

Even if there is a path from ψ_3 to ψ_4 via β (i.e., some of the states in which β is enabled in ψ_3 have successors in ψ_4), it does not necessarily follow that β leads from ψ_3 to ψ_4. The following query may be able to check for the "leads to" relation:

```
8 ?- sfenabled(trainGate,beta,3,Q),
     h(trainGate,beta,Q,SuccessorQ),
     negsf(trainGate,4,SuccessorQ).
*** No ***
```

The `No` answer indicates that there is no state satisfying

$$(\psi_3 \wedge e_\tau \wedge \neg \psi_4^\tau),$$

from which the validity of $\{\psi_3\}\beta\{\psi_4\}$ follows [see (9)].

By contrast, the transition α_2 is enabled from ψ_3, but no successor states satisfy ψ_4. Thus we obtain

```
9 ?- sfenabled(trainGate,alpha2,3,Q),
     h(trainGate,alpha2,Q,SuccessorQ).
     negsf(trainGate,4,SuccessorQ).
Q=[1,0,1,1]
SuccessorQ=[2,0,1,1]
*** Retry *** ? y
SucccessorQ=[2,0,1,1]
*** Retry *** ? y
*** No more answers ***
```

instead of the `No` answer of the previous query.

Simultaneous substitution and negation (as in $\neg \psi_i^\tau$) is performed in CLP by

```
h(Ttm,Transition,Q,SuccessorQ),
negsf(Ttm,StateformulaNo,SuccessorQ)
```

The predicate `filterpath(Ttm)` returns a list of all paths between nodes in `M.sf` that are not self-loops. The predicate `filterpath(Ttm, File)` does the same thing, except it redirects its output to `File`. A filename must be specified for `File`, and thus `filterpath/2` is not bidirectional. If the filename starts with an uppercase letter, or if it contains any characters in it that are not letters (such as a period, comma, etc.), then it must be enclosed in single quotes.

```
12 ?- filterpath(trainGate).    % find all paths for all the
                                % state formulas in trainGate.
Ttm: trainGate                  % Direct all output to screen.
From node: 0
Transition: alpha1
To node: 2
Ttm: trainGate
From node: 0
Transition: alpha1
```

```
To node: 1
:
Ttm: trainGate
From node: 4
Transition: delta
To node: 0
*** Yes***

13 ?- filterpath(trainGate, 'trainGate.paths').
                % Same as filterpath/1,
                % except output is
                % redirected to the file
                % trainGate.paths.
*** Yes***
```

Thus the `filterpath` predicate can be used to check that all the paths in the proof diagram are correct, and none have been left out.

5. DISCUSSION

In this paper we have shown that CLP($\Re$) is a useful language for representing discrete event processes and reasoning about them. The implicit symbolic nature of arithmetic constraints used in CLP($\Re$) is an important improvement over state bindings produced by unification in PROLOG, especially when dealing with infinite-state transition systems. It was argued that the operational nature of transition systems favors the use of logic programming over the use of straight theorem provers for mechanizing verification. A general theorem prover may prove more powerful if transition systems are axiomatized, but then the visual intuition of transition systems is lost.

As mentioned in the introduction, the verification techniques discussed in this paper have been applied to larger systems than the train-gate example. However, a possible area of future research is the investigation of modular decomposition methods so as to deal with much larger systems that can currently be accommodated. Another major area of research is the extension of the query module to handle liveness and real-time response specifications, which may also be verified via proof-diagram analysis.

APPENDIX A. THE CLP CODE FOR THE QUERY MODULE `QM`

This appendix mentions a few implementation details as well as giving the CLP code for the query module.

Until inequations are implemented in CLP, the following definition can be used:

```
?- op(40,xfx,neq).
A neq B :- A >= B+1; A <= B-1.
```

The clause `A > B, A < B` could have been used to define the inequation. However, since CLP has been developed for real numbers, some accommodation must be made for integer types. If there is no real solution set, then clearly there is

no integer set. Nevertheless, an answer constraint may have a real solution space, yet fail to have an integer solution space. In order to ensure that a `Yes` answer from CLP means that there is not only a real solution but also an integer solution with the constraints, it is advisable never to use the symbol < or > in simple constraints. In this way, constraints such as `| ?- 2 < X, X < 3` will not yield the answer `Yes` where in fact no integer solution exists. Obviously, any constraint `X < Y` can be transformed into `X <= Y - 1` for integer arithmetic (and a similar transformation can be used for >). For constraints where this is not enough (e.g. nonlinear constraints), queries such as `sfenabled/5` must be called instead of `sfenabled/4` so that the satisfying constraints can be checked further for integer solutions. There is thus obviously a great need for mixed integer and real constraint logic programming.

CLP($\Re$) can be improved from the user's point of view. In addition to features such as tail-recursion optimization, efficient garbage collection, and the ability to interface with other languages found in industrial-strength PROLOGs, implementation of "inequation" solving and provision of measures of numerical sensitivity would be useful.

The CLP code for `QM` is provided below:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                                                          %%%%
%%%%       general.clpr - general utilities file used by QM                   %%%%
%%%%                                                                          %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?- reconsult( '/usr/lib/clpr/all '). % get the ALL predicate. Similar
% to setof/findall/bagof

% some general predicates written in CLP.
?- op(50, xfx, :). % 'replace ' (in transformation function) %
?- op(40, xfx, neq). % 'NotEquals ' for integers. %
?- op(40, xfx, in). % eg. 3 in [1,2,3,4]. %
?- op(40, xfx, nin). % 'NotIn ' eg. 1 nin [2,3,4]. %
% See predicate below %

X in [X|Rest]. % found X in set. %
X in [Y|Rest]:- % X not found, look for next %
X in Rest. % occurrence. %

X nin [A|Rest]:- % If X neq A then check if X nin %
X neq A, % rest of set. %
X nin Rest.
X nin []. % base case: X is not in empty set %

% neq (\== in C-Prolog) for integers.
A neq B:-
A>=B+1; A<=B-1.

translateListToTerm([], true).
translateListToTerm([L|Rest], (L, Term)):-
translateListToTerm(Rest, Term).

writeList([]).
writeList([H|T]):-
writeln(H), writeList(T).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                                                      %%%%
%%%%                          QM  -  Query module                         %%%%
%%%%                                                                      %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?- reconsult( 'general.clpr ').

% map/2 to be used in conjunction with map/3
map(Esm, Order):-
        map(Esm, _, Order).

% return the required ordering of the ESM variables.
%       ie. the order they must be in for the state vector Q
ordering(Esm, Order):-
        once(map(Esm, Order, _)).

% Find all transitions ``Transition'' that are enabled from a state-
formula
%       SF for the stated ESM, (both self-loop and exiting).
sfenabled(Esm, Transition, SF):-
        map(Esm, Q),
        sf(Esm, SF, Q),
        en(Esm, Transition, Q).
% same as sfenabled/3 but returns the state vector Q as well.
sfenabled(Esm, Transition, SF, Q):-
        map(Esm, Q),
        sf(Esm, SF, Q),
        en(Esm, Transition, Q).

% Find all transitions Transition that exit state-formula SF
%       (for a given Esm) that aren't self loops.
sfexits(Esm, Transition, SF):-
     sfenabled(Esm, Transition, SF, Q),
     h(Esm, Transition, Q, NQ),
     negsf(Esm, SF, NQ).
% same as sfexits/4, but returns Q as well
sfexits(Esm, Transition, SF, Q):-
     sfenabled(Esm, Transition, SF, Q),
     h(Esm, Transition, Q, NQ),
     negsf(Esm, SF, NQ).

% find a path from state-formula SF1 to state-formula SF2 via Transition
%       for a given Esm.
path(Esm, SF1, Transition, SF2):-
        sfenabled(Esm, Transition, SF1, Q),
        h(Esm, Transition, Q, NQ),
        sf(Esm, SF2, NQ).
% same as path/4, but returns the satisfying Q as well.
path(Esm, SF1, Transition, SF2, Q):-
        sfenabled(Esm, Transition, SF1, Q),
        h(Esm, Transition, Q, NQ),
        sf(Esm, SF2, NQ).

% find ALL paths for all the state-formulas of a given Esm.
% filterpath filters out all self loops, and prints the results to
% the screen in a cleaner format.
```

```
filterpath(Esm):-
        path(Esm, SF1, Transition, SF2),
        filter(Esm, SF1, Transition, SF2),
        fail.
filterpath(_).

filter1(Esm, SF1, Transition, SF2):-
        (SF1 = SF2;
     printf(''\nEsm: %s\nFrom node: %d\nTransition: %s\nTo node: %d\n'',
                    [Esm, SF1, Transition, SF2])
        ), !.
% same as filterpath/1, except it redirects the output to File.
filterpath(Esm, File):-
        tell(File),
        path(Esm, SF1, Transition, SF2),
        filter1(Esm, SF1, Transition, SF2),
        fail.
filterpath(_, _):- told.

% Get all the transitions Transitions that exit the state-formula SF but
%       don't go to another state-formula within the predicate space.
exits(Esm, Transition, SF):-
        sfenabled(Esm, Transition, SF, Q),
        h(Esm, Transition, Q, NQ),
        metanegsf(Esm, NQ).
% same as exits/3, except it returns the constraints of the
%       satisfying state vector Q as well.
exists(Esm, Transition, SF, Q):-
        sfenabled(Esm, Transition, SF, Q),
        h(Esm, Transition, Q, NQ),
        metanegsf(Esm, NQ).
```

APPENDIX B. NEGATIONS OF STATE FORMULAS

The negation `negsf` of each `sf` predicate is given by

```
negsf(trainGate,0,[X1,X2,X3,Y,Z]):-
        X1 neq 0; X2 neq 0; (X3 neq 0,(X3 neq 1; Y=1)).
negsf(trainGate,1,[X1,X2,X3,Y,Z]):-
        X1 neq 1; X2 neq 0; X3 neq 1; Y=1.
negsf(trainGate,2,[X1,X2,X3,Y,Z]):-
        X1 neq 1; X2 neq 0; X3 neq 0.
negsf(trainGate,3,[X1,X2,X3,Y,Z]):-
        X1 neq 1; X2 neq 0; X3 neq 1; Y neq 1.
negsf(trainGate,4,[X1,X2,X3,Y,Z]):-
        X2 neq 1; X3 neq 2.
metanegsf(trainGate,[X1,X2,X3,Y,Z]):-
        negsf(trainGate,0,[X1,X2,X3,Y,Z]),
        negsf(trainGate,1,[X1,X2,X3,Y,Z]),
        negsf(trainGate,2,[X1,X2,X3,Y,Z]),
        negsf(trainGate,3,[X1,X2,X3,Y,Z]),
        negsf(trainGate,4,[X1,X2,X3,Y,Z]).
```

The negations are produced by running the UNIX tool *lex* on `trainGate.sf`. *lex* appends to the field negated predicates and `metanegsf`. The negations are used by `QM`, and cannot be done in CLP because negation is unsafe.

The author is indebted to both Jean-Louis Lassez and Joxan Jaffar for their advice on the pragmatics of constraint logic programming, as well as to the anonymous referees of this paper. Jeff Klein helped to produce the CLP code for the train-gate and numerous other examples.

REFERENCES

1. Auernheimer, B. and Kemmerer, R. A., RT-ASLAN: A Specification Language for Real-Time Systems, *IEEE Trans. Software Engrg.* SE-12(9):879–889 (Sept. 1986).
2. Abadi, M. and Manna, Z., Nonclausal Temporal Deduction, in: R. Parikh (ed.), *Logics of Programs*, Lecture Notes in Comput. Sci. 193, Springer-Verlag, 1985, pp. 1–15.
3. Bernstein, A. and Harter, P. K., Proving Real-Time Properties of Programs with Temporal Logic, in: *Proceedings of ACM SIGOPS 8th annual ACM Symposium on Operating Systems Principles*, Dec. 1981, pp. 1–11.
4. Brand, K. P. and Kopainsky, J., Principles and Engineering of Process Control with Petri Nets, *IEEE Trans. Automat. Control* 33(2):138–149 (Feb. 1988).
5. Boyer, R. S. and Moore, J. S., *A Computational Logic Handbook*, Academic, 1988.
6. Colmerauer, A., Opening the Prolog-III Universe, *Byte Mag.* 12, No. 9, (Aug. 1987).
7. Harel, D., Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Programming* 8:231–274 (1987).
8. Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, Mass., 1989.
9. Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall, 1985.
10. Hooman, J. and Widom, J., A Temporal Logic Based Compositional Proof System for Real-Time Message Passing, Technical Report TR-88-919, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., June 1988.
11. Jaffar, J. and Lassez, J.-L., Constraint Logic Programming, in: Proceedings of ACM Symposium on Principles of Programming Languages, *Munich, Jan.* 1987.
12. Jahanian, F. and Mok, A. K., Safety Analysis of Timing Properties in Real-Time Systems, *IEEE Trans. Software Engrg.* SE-12(9):890–904 (Sept. 1986).
13. Koymans, R., Bytopil, J., and de Roever, W. P., Real-Time Programming and Asynchronous Message Passing, in: *Proceedings of the 2nd Annual Symposium on Pirnciples of Distributed Computing*, Montreal, Aug. 1983, pp. 187–197.
14. Kramer, J., Magee, J., and Sloman, M., A Software Architecture for Distributed Computer Control Systems, *Automatica* 20(1):93–102 (Jan. 1984).
15. Koymans, R., Shyamasundar, R. K., de Roever, W. P., Gerth, R., and Arun-Kumar, S., *Compositional Semantics for Real-time Distributed Computing*, Lecture Notes in Comput. Sci. 193, Springer-Verlag, June 1985.
16. INMOS Limited, *Occam Programming Manual*, Internat. Ser. Comput. Sci., Prentice-Hall, Englewood Cliffs, N.J., 1984.
17. Leveson, N. G. and Stolzy, J. L., Safety Analysis Using Petri Nets, *IEEE Trans. Software Engrg.* SE-13(3):386–397 (Mar. 1987).
18. Lee, I. and Zwarico, A., Timed Acceptances: A Model of Time Dependent Processes, in: M. Joseph (ed.), *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Comput. Sci. 331, Springer Verlag, 1988, pp. 36–66.

19. Menasche, M., PAREDE: An Automated Tool for the Analysis of Time(d) Petri Nets, in: *International Workshop on Timed Petri Nets*, IEEE Comput Soc., June 1985, pp. 162–169.

20. Milne, G. J., CIRCAL and- the Representation of Communication, Concurrency and Time, *ACM Trans. Programming Languages and Systems* 7(2):270–298 (Apr. 1985).

21. Magee, J., Kramer, J., and Sloman, M., Constructing Distributed Systems in Conic, *IEEE Trans. Software Engrg.* 15(6):663–675 (June 1989).

22. MacEwen, G. H. and Montgomery, T. A., Expressing Requirements for Distributed Real-Time Systems, in: *Abstracts of the IEEE Computer Society: 4th Workshop on Real-Time Operating Systems*, July 1987, pp. 125–128.

23. Moszkowski, B., A Temporal Logic for Multilevel Reasoning about Hardware, *Computer* 18(2):10–19 (Feb. 1985).

24. Manna, Z. and Pnueli, A., How to Cook a Temporal Proof System for Your Pet Language, in: *Proceedings of the Symposium on Principles of Programming Languages*, Austin, Tex., Jan. 1983, pp. 141–154.

25. Merlin, P. M. and Segall, A., Recoverability of Communication Protocols—Implications of a Theoretical Study, *IEEE Trans. Comm.*, Sept. 1976, pp. 1036–1043.

26. MacEwen, G. H. and Skillicorn, D. B., Using Higher-Order Logic for Modular Specification of Real-Time Distributed Systems, in: M. Joseph (ed), *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Comput. Sci. 331, Springer-Verlag, 1988, pp. 36–66.

27. Narayana, K. T. and Aaby, A. A., Specification of Real-Time Systems in Real-Time Temporal Interval Logic, in: *Proceedings Real-Time Systems Symposium*, IEEE Computer Soc., Dec. 1988, pp. 86–95.

28. Ostroff, J. S., *Temporal Logic for Real-Time Systems*, Adv. Software Development Ser., Research Studies Press (distributed by Wiley), England, 1989.

29. Ostroff, J. S., Deciding Properties of Timed Transition Models, *IEEE Transactions on Parallel and Distributed Systems*, Volume 1, No. 2, pages 170–183, April 1990.

30. Ostroff, J. S. and Wonham, W. M., Modelling, Specifying and Verifying Real-Time Embedded Computer Systems, in: *Proceedings of the 8th IEEE Real-Time Systems Symposium*, San Jose, Dec. 1987, pp. 124–132.

31. Ostroff, J. S. and Wonham, W. M., A Framework for Real-Time Discrete Event Control, *IEEE Trans. on Automatic Control*, Volume 35, No. 4, pages 386–397, April 1990.

32. Pnueli, A., The Temporal Logic of Programs, in: *Proceedings of the 18th IEEE Annual Symposium on the Foundations of Computer Science*, Providence, Nov. 1977, pp. 46–57.

33. Pnueli, A., Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends, in: J. de Bakker, W. P. de Roever, and G. Rozenburg (eds.), *Current Trends in Concurrency*, Lecture Notes in Comput. Sci. 244, Springer-Verlag, 1986.

34. Quirk, W. J., *Verification and Validation of Real-Time Software*, Springer-Verlag, Berlin, 1985.

35. Ramchamdani, C., Analysis of Asynchronous Concurrent Systems by Timed Petri Nets, Technical Report MAC TR 120, MIT, Feb. 1974.

36. Razouk, R. R. and Phelps, C. V., Performance Analysis of Timed Petri Nets, in: *Proceedings of 4th International Workshop on Protocol Verification and Testing*, June 1984.

37. Reed, G. M. and Roscoe, A. W., A Timed Model for Communicating Sequential Processes, in: *Proceedings ICALP 86*, Lecture Notes in Comput. Sci. 226, Springer-Verlag, 1986.

38. Stankovic, J. A., Misconceptions about Real-Time Computing: A Serious Problem for Next Generation Systems, *Computer* 21(10):10–19 (Oct. 1988).

39. USDOD, *Reference Manual for the Ada Programming Language*, Springer-Verlag, New York, 1983.
40. Voda, P. J., Types of Trilogy, in: R. A. Kowalski and K. A. Bowen (eds.), *Logic Programming: Proceedings of the 5th International Conference and Symposium*, MIT Press, Cambridge, Mass., 1988, pp. 580–589.
41. Wirth, N., Towards a Discipline of Real-Time Programming, *Comm. ACM* 20, No. 8 (Aug. 1977).
42. Zave, P., An Operational Approach to Requirements Specification for Embedded Systems, *IEEE Trans. Software Engrg.* SE-8(3):250–269 (May 1982).
43. Zubrek, W. M., Timed Petri Nets and Preliminary Performance Evaluation, in: *Proceedings 7th Annual Symposium on Computer Architecture*, La Baule, France, 1980.